Making Patterns Explicit with Metaprogramming

Daniel von Dincklage

University of Colorado at Boulder, Boulder CO 80309, USA
danielvd@cs.colorado.edu

Abstract. Design patterns have been a useful tool for a better understanding of the collaboration between several classes and objects in a program.
One drawback of this approach is the lack of an explicit representation of the patterns used in a program, as the collaboration between classes is normally expressed in the code of the class itself.
In this paper, we present a method for explicitly representing patterns in a program with the help of metaprogramming techniques. The method presented has benefits compared to traditional approaches with respect to documentation and reusability of the program, as well as providing a better separation of the protocol contained in the pattern.

1 Introduction and Motivation

Design patterns are frequently used as a tool to guide and document the design of object-oriented software systems. The benefits of the use of design patterns are broad, as they can be used to abstract from low-level interactions between classes and provide flexible solutions to common problems.

The specification of a design pattern is usually done using a particular template that includes a number of elements required to express the design intent that is to be captured. One of the elements of the specification usually is the classes involved - the approach taken here is to label each class with the role it plays in the whole collaboration. Also specified is the form of collaboration between the participants as required by their specific role. Other elements are quite important in a pattern too - especially the "soft" issues, such as the motivating design problem or design tradeoffs to be made when employing the pattern.

Another part of the pattern is obviously concerned with the method of actual implementation. Often a pattern description will contain some examples and a discussion of issues which are concerned with the implementation phase itself, such as the peculiarities of specific programming languages.

These issues during the implementation of patterns have been analyzed by several authors [1,2,3,4]. One of the most common conclusions is that the language or paradigm used affects the implementation effort. It has even been observed that patterns can even "disappear" in the language - the actual use of the pattern becomes completely transparent, as described in [2,3]. One interesting observation is that often dynamic languages provide the possibility of very simple implementations of design patterns.

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 287–306, 2003.
© Springer-Verlag Berlin Heidelberg 2003

In contrast to these previous approaches the main goal of this paper is to provide an analysis of the benefits of static metaprogramming when applied to pattern implementation as a single language feature rather than a whole language, as done in [1].

The focus on the pattern "implementation" is also weighed in differently: One issue with design patterns is that they sometimes involve quite large implementation burdens that are largely trivial wiring between classes. An example of this is the Proxy pattern:

For each of the methods present in the public interface of the object proxied, the proxy itself must also provide the same method, with an almost trivial implementation. Writing these routines manually can be very tedious and error-prone, while also complicating maintainance.

To alleviate this problem is one of the goals of this paper. The idea behind the approach is to use a programming technique called Metaprogramming. Metaprogramming is understood as the concept of creating programs that will themselves produce other programs, or manipulate other programs according to their algorithms. Of course, due to this very broad definition there are quite a number of systems that can be classified as metaprogramming systems. Due to this, metaprogramming systems have been categorized with respect to a number of traits [5]. In this paper, the metaprogramming system used is a static one. Static metaprograms are run at compile time of the subject program. They are invoked with the original source program and transform it into another source-level program. The actual nature of the transformation is of course dependent on the metaprogram, as it could insert or delete program statements, procedures or whole classes. These capabilities are important for the goal of this paper - to produce metaprograms that can be used as a form of reusable patterns. As described earlier, it can be very tedious and error-prone to actually implement some of the patterns, work, which could be partially automated. Later, the user only would have to supply the metaprogram with the class the pattern should operate on, and supply some pattern-specific data. This would then be expanded by the metaprogram into the actual code which contains the implemented pattern. This would not only have the benefit of vastly speeding up development and introducing fewer bugs, but would also have advantages with respect to the expressiveness of the code, and thus the documentation as well as making the program more resistant to change. Previously, as the pattern was made up of a lot of hand-written implementation, it was sometimes not necessarily obvious that or which pattern had been used. In contrast to that, if the use of a pattern is expressed in the application code by an explicit invocation of a metaprogram, which receives the important properties of the pattern as parameters, an explicit documentation is present in the code itself.

Although the transformations described in this paper are quite simple and could be performed with any sufficiently expressive metaprogramming system, the system chosen for the actual implementation of the system is Common Lisp with a full object system and meta object protocol [6], as it has very easily usable metaprogramming capabilities.

The main problem with implementing a similar system with other programming languages is that the approach taken to implement the patterns most closely resembles the extension of the original source language with new constructs in such a way that these seamlessly integrate into the original language. These new constructs are then used by the programmer by specifying the appropriate metadata in order to invoke them.

Unfortunately, this prevents the use of purely dynamic reflection systems, such as present in Java, as there is no easy way to communicate the metadata specified by the programmer to the dynamic systems, provided there is no specific support for adding metadata to programs (such as in Eiffel or languages based on the MSIL, such as C#).

However, if there is the possibility to introduce these extensions, such as using a system like ELIDE [7], there is no specific limitation that would prevent the use of languages other can Common Lisp. Especially unproblematic is the use of statically typed languages.

The idea of expressing patterns as meta-programs obviously leads to another question - that of behaviorally similar patterns. As in [8] and similar work, patterns usually have been classified according to their intent, as it is seen by the implementor. This clearly has advantages when choosing the right pattern for a given problem, as each pattern is specifically adapted to its special usage. However, this fails to be helpful when trying to provide the implementations of patterns as metaprograms, as this conventional classification does not provide a taxonomy that is useful for the reuse of the pattern implementations.

2 The Benefits of Metaprogramming for Patterns

As already noted in [2,3], for some of the GoF-patterns dynamic languages are very appropriate. The best example for this property is some of the creational patterns - As noted there, in some dynamic languages the implementation of these patterns degenerates unto the point where it disappears. These easily implementable patterns are (in GoF-terms) Abstract Factory, Prototype and Singleton

In order to generate a reusable implementation of these patterns with the help of metaprogramming, a possible approach is to create a new metaclass that implements the instantiation protocol specified by the respective pattern. The pattern can then simply be reused by declaring a new class as an instance of the metaclass associated with the pattern.

To now implement the Singleton pattern it is advisable to first analyze the pattern itself. The idea of this pattern is to limit the number of instances of a particular class to one . Usually this is implemented with the help of a language feature - “static” functions in Java or “once” in Eiffel. The actual protocol of the pattern is quite simple: The user calls a predefined function, it then will return the unique instance.

An implementation using CLOS is somewhat different, as it neither uses a special language feature to control the number of instantiations, nor requires a distinct function to return the instance, as shown in listing 1.

```
(defclass singleton-class (standard-class) ((store ::initform nil)))

(defmethod allocate-instance ((class singleton-class) &rest args)
(if (null (slot-value class 'store))
  (setf (slot-value class 'store) (call-next-method class args))
  (slot-value class 'store)))

(defclass shape () () (:metaclass singleton-class))
(make-instance (find-class 'shape))
```

Listing 1: The singleton example

One observation is that the actual code implementing the "singleton" behavior can be expressed regardless of the clients that later use it, as well as the fact that a client adheres to the "singleton"-protocol of instantiation is made explicit by the `:metaclass`-specification during class definition. Providing such explicit indications of the usage of patterns is the main goal of this paper.

The two other patterns mentioned - Abstract Factory and Prototype are quite similar with respect to their behaviour. In both cases, in order to abstract from the concrete instantiation process, methods are called that will return the instance that should be used by the client. The object instantiation itself is done according to some internal algorithm inside of the called method. Implementing these patterns would be similar to the implementation of the singleton.

One interesting property of these three patterns is that they exhibit a striking similarity between each other, up to a point where all three can be regarded as specializations of a more general pattern "creator" that exposes the instantiation protocol of the programming language used, and thus spans multiple abstraction levels.

With the notion of a more general "Creator" pattern the reason for the simple and similar implementations in dynamic languages of the three patterns mentioned becomes obvious:

As the normal object instantiation protocol is also a specialization of the more general pattern, this special instance of the Creator pattern must already be implemented in every object-oriented language. The advantage of dynamic languages, however, is that exactly this implementation is exposed and easily available to the programmer, allowing the programmer to modify it.

The difficulties that arise when implementing these patterns in more static languages also become understandable - there, the instantiation protocol is usually implemented in the compiler, and thus out of reach of the normal application programmer.

With this example, the advantages of implementing patterns with the help of metaprogramming should have become obvious - both with respect to reusability, documentation and separation of code, as well as conceptual benefits.

3 Devising Metaprograms

As shown in the previous section, it is desirable to have the language already implement parts of the design pattern, as it can save considerable overhead while at the same time producing better code.

Of course, it is an important question how these metaprograms actually work. The idea is that the user can specify the application of the design pattern with a special syntax in his original program. While specifying the application, the user also supplies metadata to the static metaprogram, such as the classes involved in the design pattern.

The benefit of this approach is that, afterwards, the pattern is made explicit in the code, as it is represented in the invocation of the metaprogram. Ideally, the result of the metaprogram itself is another class that then can be processed by additional metaprograms. At first glance, the results of the approach might seem different from the approach taken by the language designers for implementing the Creator pattern, but it certainly is not: The difference is in the actual method of parameterizing the pattern. Whereas the Creator pattern is parametrized implicitly by overloading specific functions, the metaprograms are parametrized explicitly at the time of invocation. The benefit of the latter method is that the parameters are made explicit as such, whereas the former requires knowledge about the instantiation protocol.

A very important part of the actual process is the method used to create the metaprograms. As already mentioned, the metaprograms receive classes and data relating to the pattern, and with that information generate additional behavior in the classes. However, in order to have a useful metaprogram it is desirable to supply it with parameters that express a great deal of information while imposing very little burden on the user; the more information the metaprogram can deduce by itself, the more useful it is. Thus, it is important not to require too much parametrization. A metaprogram that takes two pieces of code, only to execute them in a given order, provides too little benefit to the user to be accepted.

Thus, the question of how a metaprogram can deduce as much information as possible becomes quite important. One key is the actual mode of collaboration used by the participants of the implemented pattern. The more specific this protocol of interaction is, the more data can be deduced by the pattern and thus automatically be implemented. Due to this, it is obvious that the main benefit will be achieved by patterns that rely heavily on objects collaborating according to a strict protocol, whereas patterns that only loosely define interfaces that can vary significantly in terms of behavior will exhibit only minor improvements.

4 The Mediator Pattern

A frequent task handled by a pattern is dispatching requests that are directed by a sender to a number of clients, thus performing the function of a mediator. A central mediator object receives messages from attached objects and processes

these messages according to rules and possibly its internal state . Resulting messages are then distributed to the associated objects. Replies are collected and returned to the original sender.

Implementing the Mediator pattern as a reusable piece of software therefore requires three fundamental sets of data that have to be supplied in order to generate a usable mediator class:

- The set of messages the mediator must accept. Each declaration of a message consists of an "uplink" and a "downlink" format. These formats specify the data transmitted in the message. The first format is used for transmission from a colleague or client to the mediator, the second format for transmission from the mediator to the colleagues,
- A set of rules associated with each message to perform the message-processing internal to the mediator and to determine the set of associated objects that should receive the result of the computation by the mediator. If the two formats are different, these rules also have to handle the transformation between uplink- and downlink-format,
- in the case of statically typed languages, the basetypes of the associated objects.

With this information, one can generate a mediator class that will adhere to a specific mediator protocol specified by the user. The meta-program to transform a class into a mediator-class adhering to a specific protocol would work somewhat like this:

- add a new variable to the class to store the colleagues at runtime,
- for each message specified, generate a function with a name specific to the message, accepting messages in the uplink-format, that will:
  - perform optional transformations of the content of the message,
  - filter the list of colleagues according to some internal criteria,
  - distribute the message to the resulting colleagues by invoking a method with a name specific to the message and pass the contents of the message in the appropriate downlink format to it,
  - If the message has a return value, apply a fold function to the results of the clients, in order to generate a possibly distilled version of all return values
- store the properties of each message as a static item associated with the class defined, ideally as part of its type.

Defining the colleague classes corresponding to the protocol that one specific mediator-class adheres to is done in this fashion:

- add a new variable to the class that will store a reference to objects of the type of the corresponding mediator-class,
- for each message defined by the mediator class generate a method, with the name being the one chosen by the mediator as "receiver" name, accepting messages in the downlink format. This method either will discard the message received or forward it to another function specified by the user, in order to be processed,

```
(defmediator the-mediator ()
  ((some-value :accessor some-value :initarg :some-value))
  (:message (:name message-a) (:format (one-arg))
    (:distributor (lambda (x y)
      (print (format nil "Received Message. Data: ~S" y)) y))))

(defmediator-client the-mediator-client ()
  ((color :accessor color :initarg :color :initform 'clear))
 (:mediator the-mediator)
 (:receive message-a #'message-a-handler)
 (:send message-a the-uplink)
 (:autoregister))
```

Listing 2: The mediator example

– for each message specified to be sent from the class to the mediator, define a method to handle that distribution.

This approach will obviously enhance the actual typesafety of the program, as each message is explicitly represented by a set of collaborating methods. Another advantage is the reliance on metadata - the programmer only supplies the messages together with a couple of specific functions. The metaprogram then wires together the logic needed to perform the communication.

Using the implementation of the pattern, a specification of a mediator would look like the one in listing 2 : The mediator shown there understands a message to communicate with clients - "message-a". It is associated with a specific translation procedure that prints the content of the message and returns it to distribute it to all clients. The client is specified to communicate with mediators of type "the-mediator" and, upon instantiation, autoregister with the then specified instance of the mediator itself. As for communication the client announces that it wants to receive messages of type "message-a" with a specific handler and ignore other messages. The client also specifies that it desires to distribute the message by calling its instance method "the-uplink".

An important thing to note is the benefit gained from implementing the Mediator pattern with metaprogramming. Of course, one could implement the pattern with the help of normal classes, but this would incur a design decision - Either the protocol between the clients and the main mediator class is based upon methods or on more dynamic message exchange: While the former approach has the advantage of being typesafe, it is much less maintainable, as new messages require extensive modification. The maintainance issue is not present in the alternative approach, which, on the other hand, is not typesafe. This situation forces the designer to make a decision upfront, which later on could prove problematic.

This problem is not present if the metaprogramming approach is taken, as the methods used for message exchange are added at compile time and can then be checked statically. Also, the programmer does not have to care anymore about the concrete method of message exchange.

5 Specializing the Mediator Pattern

After having defined the Mediator pattern as an explicit metaprogram, it is now possible to analyze different patterns with respect to their similarity in communication protocol with the mediator pattern. When doing this analysis for the GoF-patterns, a large group of patterns can be found to be substantially similar:

When regarding other patterns as specialized versions of the Mediator, it becomes obvious that most of them specialize the Mediator pattern in four different ways:

- **Limiting the direction of communication**
  Many patterns specialize the mediator protocol in such a way that the communication is unidirectional from the mediator to the clients.
  Examples: Observer, State, Decorator, Composite, Adapter, Bridge, Proxy, Chain of Responsibility
- **Limiting the number of clients**
  A number of patterns exhibits a similar communication protocol as Mediator, but limits the number of clients to 1. These patterns usually define the same interface for the mediator and the single client.
  Examples: Adapter, Bridge, Proxy, Decorator, Chain of Responsibility
- **Limiting the message format**
  Frequently patterns define a fixed set and format of messages. This set is usually fixed to a subset of or the whole external interface of the mediator, creating a 1:1 relation between the external methods of the mediator and supported messages.
  Examples: State, Decorator, Chain of Responsibility, Bridge, Proxy, Composite
- **Defining the "distribution selection" method**
  Some patterns specialize the Mediator pattern in a way that they define the method used to select which clients receive messages.
  Examples: State, Chain of Responsibility, Adapter, Bridge, Proxy

5.1 Specializing the Mediator Pattern for "Adapter"

The intent of the "Adapter" design pattern is to offer clients an expected interface, while implementing the functions it contains with the help of a class that exposes a different interface.

The protocol used here is very similar to the one of the mediator: The adapter receives a message from a client, transforms it according to internal rules into one or more messages that then are sent to the adaptee[1].

In fact, the main task of the Adapter pattern is the actual translation between the signature of a method offered by the adapter, and the signature of the corresponding method of the adaptee.

[1] We only consider object adapters, as class adapters are merely a performance enhancement.

```
(defclass window-impl () ())
(defclass x11-impl (window-impl) ())
(defclass pm-impl (window-impl) ())

(defbridge window () () (:target window-impl)
 (:bridge (:from drawline) (:to impl-line))
 (:bridge (:from drawtext) (:to impl-text)))
```

Listing 3: The bridge example

As this process corresponds to the translation between an "uplink"-message-format (≡ the signature of the method in the adapter) and a "downlink"-message-format (≡ the signature of the method in the adaptee), everything important is already provided in the implementation of the Mediator pattern.

5.2 Specializing the Mediator Pattern for "Bridge"

The intent of the Bridge pattern is to separate concrete implementations of abstractions from the refinements of the abstractions themselves. In an actual implementation this is usually done by constructing two different class-hierarchies, one for the abstractions, and one for the implementation. As the behavior of the refined abstractions is defined in terms of the more general abstractions, it is sufficient for the implementations to implement an interface equivalent to the most general abstraction provided.

While the conceptual idea of the Bridge pattern is somewhat complicated, the actual protocol it adheres to is not: The most general abstraction maintains a reference to a concrete implementation object to which it will forward "primitive" messages.

Expressing this protocol as an instance of the mediator-protocol is very simple - Each method bridged corresponds to one message, the format of which is the same as the arguments of the primitive method. The receiving function is the method to be invoked in the concrete implementation, the sending function the method of the abstraction.

The use of the resulting pattern is shown in listing 3, similar to the example in [8].

5.3 Specializing the Mediator Pattern for "Composite"

The intent of the Composite pattern is to enable the composition of tree structures in order to represent part/whole hierarchies.

The protocol used by this pattern involves a single Composite object that holds references to multiple parts and exports two different interfaces to its clients. The first interface is the part-interface. This interface is common to all the elements of the structure, the composite and the clients. The second interface is special to the composite object, in that it exports routines to manage

```
(defdecorator decoratorA (component) ()
 (:decorate (:name funB) (:target component)
    (:before (lambda (x y) (print "Decorating") y))
    (:after  (lambda (x y) (print "Decorating some more"))))
 (:proxy (:names funA)))
```

Listing 4: The decorator example

the multiple parts the composite holds references to. Messages received by the composite in the part-interface are distributed to all of the parts it currently holds a reference to. Additionally, the composite can execute code before or after the distribution.

The specialization of the "Mediator" pattern to limit the protocol is quite simple - One only has to limit the message filtering function to always let messages through. The system automatically should also generate a message for each routine present in the part-interface, including a sender-function in the composite object itself. As the mediator already implements the possibility of executing code before or after the message distribution, this capability can be directly reused.

5.4 Implementing "Decorator" and "Chain of Responsibility"

The use of the Decorator pattern is to provide an easy way of extending functionality of objects without subclassing these. This is implemented by providing a fixed set of operations in the objects that later on should be able to be extended. The decorators, deriving with the extendable objects from a common superclass, both provide a custom extension to these operations and forward the original request to the object of the type of the common superclass which they extend at runtime.

The protocol this pattern uses for its communication is very similar to the ones already described - The decorator awaits requests from the clients and forwards requests that it chooses not to handle itself to the object it holds as a reference. The main capability of the decorator, however, is to possibly execute code before or after the forwarding of a request, allowing the decorator to dynamically attach code to its client.

The relation of the Decorator pattern to the Composite pattern is quite obvious , the only difference being that the protocol of the Decorator pattern limits the number of parts the composite can hold references to to exactly one.

This specialization does not involve much work - the single difference would be that for the routine possibly executed after the forwarding of the request, it could be desirable to eliminate the fold semantics of the method executed after forwarding the request, as there is only a single client.

The actual usage is done similarly to the program in listing 4, which declares a decorator-class "decoratorA" that will forward function calls to the function funA, but decorate calls to the function funB.

```
(defchain-of-resp elementA (component) () (:target component)
  (:request (:name funB)
    (:handler (lambda (x y) (print "Handling request.") y))
    (:has-handled (lambda (x y z) t))))

(defchain-of-resp elementC (component) () (:target component)
  (:proxy (:names funB)))
```

Listing 5: The chain-of-responsibility example

The Chain-of-Responsibility pattern is very similar to the Decorator pattern with regard to the protocol used. Whereas in the Decorator pattern the request is always forwarded to the "main" object and the decorators attach additional responsibilities to it by executing code, in the Chain-of-Responsibility pattern the interaction is different - here, the attached objects do not execute their own code and then forward the request, but rather only forward the request and stop the forwarding in the event that they have handled the request sufficiently. This leads to a very similar implementation, that almost only differs in the terminology offered to the user, as shown in listing 5.

An obvious extension to this version of the Chain-of-Responsibility pattern would be to also allow decorations by the elements of the chain. In that case, the Decorator pattern could be regarded as a specialization of the thus modified Chain-of-Responsibility pattern.

5.5 Specializing the Mediator Pattern for "Proxy"

The Proxy pattern is used to provide a placeholder object in order to control the way how the "real" object is accessed.

Again, as in the Bridge pattern the protocol used is relatively simple: The proxy object awaits messages, and, upon receiving them, forwards them to the real object. If this protocol is implemented as a specialization of the mediator-protocol, one can use a mapping nearly identical to the one used for the Bridge pattern. This is due to the fact that these two patterns are also structurally very similar - the only major difference being the fact that with the Proxy pattern the "abstraction" and "implementation" hierarchies are put together into one common tree. The one addition very useful for the Proxy pattern is the possibility to provide a single key that will add a transformation for all the routines in a given class with the specified processing involved. This alleviates the problem of having to specify each routine by hand.

In this example there are three classes involved - "picture" as an abstract superclass for all picture-related things, "real-picture" as the concrete implemenetation and "picture-proxy" as a proxy that forwards calls to the method show.

The difference between these classes is first of all the ":target"-directive indicating the class proxied. The ":proxy"-directives are used to specify the routines that actually should be proxied, in this case show. As the actual picture should

```
(defclass picture () ())
(defmethod show ((x picture)) (error "Abstract!"))

(defclass real-picture (picture) ())
(defmethod show ((x real-picture)) (print "A nice picture."))

(let ((fun (lambda (x y)
  (unless (mediator-has-client x)
     (print "Loading picture from disk...")
     (add-client x (make-instance (find-class 'real-picture)))
     (print "Done.")) y)))

 (defproxy picture-proxy (picture) () (:target picture)
  (:proxy (:names show) (:processor fun))))
```

Listing 6: The proxy example

be loaded prior to accessing, a processing function that will do just that is neccesary. One thing to note is that the processing function needs only be defined once, as it can be used in multiple proxying rules.

One can see the ease with which a new proxy can be specified when using the specialized proxy-metaprogram. Next to this there are, of course, several other advantages - such as the improved documentation and better resistance to change. This is mostly due to the fact that the actual specification of the proxied methods is independent of their parameter declaration. A similar argument applies to the specification of the processing-function: This function does not have to be specified for each proxied function separately, and thus can be changed more easily.

5.6 Specializing the Proxy Pattern for "State"

The State pattern is used to ease the modeling of an internal state of a given object. This is done by providing a central context-class that denotes the whole object to the external world. This context-class holds a reference to a current State-object. This object is an instance of one of multiple subclasses of a common State-class. As long as the Context-object retains its state, all appropriate requests will be handled by the currently associated state object. If the state of the Context-object changes, it will replace the current State-object with an instance of a different subclass of the common State-class.

The protocol involved in this pattern is again quite simple - The central Context-object awaits requests and then forwards these to the current state object. This obviously is a simple extension of the Proxy pattern: The proxy has only one client of a set of different classes. This client frequently is exchanged by other clients, while all interesting method calls are forwarded to the current client. In addition to the behavior as defined by the proxy, there is also a sec-

```
(defsubject shape-subject ()
  ((color :accessor color :initarg :color :initform 'clear))
  (:observe (:name color)))

(defmediator-client the-observer () ()
 (:mediator shape-subject) (:receive color #'handler))

(defmethod handler ((l the-observer) slot old new ) [...] )
```

Listing 7: The observer example

ondary part that is responsible for determining the actual class of the "next" state.

The metadata specified by the user in this case would consist of three parts apart from the abstract class that all the states inherit: The first one is the set of functions that need to be forwarded from the central dispatching object to the current state. The second piece of metadata is a transition-function that describes how the states are changed. A start-state also has to be specified .

5.7 Specializing the Mediator Pattern for "Observer"

The usage of the Observer pattern is to automatically update clients of a given object, should certain properties of the observed subject change.

The protocol involved with this communication is quite simple - as soon as the observed event is recorded in the subject, the subject sends a given message to all of its clients who then react accordingly.

As a specialization of the Mediator pattern, it determines the messages exchanged between the observer and the subject by fixing the possible events the observers actually can observe. In addition to the types of messages, the format of the messages themselves is also limited, as one usually will limit the transmitted information to a reference to the subject observed and possibly the new and old value of the data observed.

The actual usage of the generated system is very simple, as demonstrated in listing 7. In this example two classes are defined: first of all the shape-subject class, which will operate as a subject. The user also specifies the actual slots of the class he wishes to observe with the help of the ":observe"-keyword. The system then internally translates this into an appropriate definition for a message of the same name as the observed slot. The system also automatically adds code that is used to determine the (write) access of the slot.

The actual usage of the message is in the definition of the client that will receive the data - here the user specifies the receiving end of the message just as in the Mediator pattern. Of course, the method attached to the message has now a predefined layout, just the same as the sender of the method specifies.

6 Other Patterns

Due to space constraints it is not possible to present an exhaustive analysis of all the GoF patterns. Nevertheless, a short overview of the properties of the remaining patterns shall be provided.

6.1 The Builder, Command, Factory Method and Iterator Patterns

These four patterns share similar problems, as implementing metaprograms for them would not prove very beneficial.

For the Builder pattern this is due to the fact that the specification of the configurable parts of the pattern (the interface of the Builder, the implementation of the Director and the concrete Builder) contains almost all of the work that needs to be done to implement the pattern.

A similiar analysis applies to the Command pattern. As the main work while implementing the Command pattern is reflected in the actions performed by the concrete command objects upon the receivers, there is no substantial benefit to be gained by implementing it as a metaprogram.

With the Factory Method pattern, the user has to supply the metaprogram with data about which subclass of the "Creator" object returns which concrete instance. This data completely covers the semantics associated with the Factory Method pattern, resulting in no real benefit apart from better documentation.

For the Iterator pattern, the intent is to provide access to a collection of objects in an abstract fashion. The main work for implementing the Iterator pattern is to map the interface of the iterator to the one of the datastructure. As datastructures vary, significant change to the iterators is required, although there have been several approaches to solve this problem, such as [9]. Unfortunatly, this leads to an insignificant amout of code that can be automatically generated by a metaprogram.

6.2 The Facade Pattern

As already described in [8], the Facade pattern is somewhat similar to the Mediator pattern. The main difference is that the facade object is not known to the object it hides. Additionally, facade objects usually implement little internal logic. These properties can be used to specify a specialization of the Mediator pattern, although the resulting metaprogram is not substantially different.

6.3 Flyweight

When implementing the Flyweight pattern, the relation of it to the Singleton pattern is apparent: For both patterns, the object instantiation plays a crucial role, as a common object is created and returned. An instance of the Singleton pattern can be regarded as a specialization of the Flyweight pattern, in that the singleton object does not require the passing of an external state to its functions.

6.4 The Interpreter Pattern

The implementation of the Interpreter pattern would be simply the same as the implementation used by the Composite pattern, as their structure is essentially equivalent. This result is similar to the one presented by Agerbo [10], who classifies the Interpreter pattern as an application of the Composite pattern.

6.5 The Memento Pattern

The Memento pattern stores the internal state of an object. Despite a simple protocol, one can experience benefits by using a metaprogram. The metadata consists of a description of the partial state of the object the memento object should be created for. From this data, the metaprogram can generate code to save and restore the particular state[2].

6.6 The Patterns Strategy and Template Method

These two patterns are very similar - Both delegate part of the actual work they do to abstract functions. The main work an implementor does for these patterns is reflected in both the interface and the implementations of the different concrete parts of the pattern. As there is no substantial protocol involved that can be derived from metadata, the benefit of using a static metaprogram to implement these patterns is very limited.

6.7 The Visitor Pattern

The Visitor pattern lends itself very well to an implementation as a metaprogram, as its protocol is defined very narrowly. In fact there have been multiple implementations of automatically generated Visitor patterns or similar applications in various systems, such as DJ [11].

7 Related Work

There are multiple areas in which related work has been produced. In particular, the areas work applicable to the material presented in this paper are:

- The creation of pattern libraries or tools to handle design patterns explicitly
- The impact of language features or paradigms upon the effort required to implement patterns
- The classification of patterns, in order to identify relationships between them

[2] With the same metadata, one could also generate copy- and comparison-functions.

7.1 Pattern Libraries and Tools

The most direct relation to the work presented in this paper is contained in pattern libraries and tools to handle the implementation of patterns.

Agerbo [12] presents a library of classes that implements their fundamental design patterns and is used by inheriting the library classes. As the pattern itself is reused by simply inheriting it, there is no possibility for the implementation of the pattern to adapt statically to the context of its use by examining the classes operated upon. This prevents any substantial code generation when compared to the approach taken in this article.

In his work, Soukop [13] introduces pattern classes. These classes act as templates that contain the methods used for the specific role of an object in the given pattern. The pattern classes then are woven together with the data contained in the application classes by a special program. As the weaving process can only perform very limited transoformations, such as the introduction of a piece of code that cannot be adapted by information, such as the names of the methods, found in the transformed class, an effective generation of code for many patterns, as presented in this article is not possible.

Tokuda and Batory [14,15] describe a set of refactorings, which includes the application of a certain set of design patterns. These refactorings can be applied by a user to an existing program which will subsequently be transformed automatically to reflect the explicit occurence of the particular pattern. As this process is mainly concerned with the discovery of design pattern and a subsequent transformation in a more maintainable representation, the relation to the material presented here is somewhat indirect.

The ELIDE System [7] allows the user to introduce additional parametrized modifiers into Java programs. These modifiers are associated with program transformations which can be used to attach code or otherwise modify the initial program. Using this technique, the user is able to implement transformations similar to the ones presented in this article.

Another system was developed by Budinsky [16]. This system provides a web-based user interface that allows different design choices to be made that are appropriate to the design pattern. The system then transforms templates for each design pattern and produces code that then can be integrated into the final application by the user. However, the transformations that can be applied to the various patterns by the system are extremely limited in that they cannot depend on properties of the code which is to be extended by the particular pattern. This severely limits the possible benefit as only very schematic code can be produced.

Alexandrescu [17], with Vlissides in [18] present a technique using C++ templates and inheritance to provide the user with parametrizable patterns. The configurable parts are encoded as template parameters and are resolved during compilation by the C++ compiler. However, as no compile-time access to the structure of manipulated classes is possible, the expansion cannot deduce information from them, forcing the user to configure all aspects.

Bosch [19] describes an object model called LayOM that implements patterns by composing objects of a number of different layers that filter the messages re-

ceived by the object. Patterns are implemented by defining layers which then can be attached to application classes. While also presenting patterns as first-class entities that can be used by the programmer, the implementation of the pattern depends on the capabilites of the object model provided, thereby limiting the extent of configuring the particular pattern implemented. This is also reflected in problems that arise if one were to implement patterns as specializations of others in the model presented by the author.

7.2 Design Patterns and Language Properties

Work in this area indirectly relates to the material presented in this paper. This is due to the fact that the approach taken here relates to the use of a single feature, static metaprogramming.

Baumgartner, Laufer and Russo [1] analyze implementations of Patterns in common programming languages in order to determine a set of single language constructs that ease the implementation of design patterns. Major constructs determined to be useful include separation of types and classes, lexically scoped closure objects, metaclass objects as well as multiple dispatch.

Norvig [2] and Sullivan [3] both analyze the effects of dynamic languages upon the effort required for implementing design patterns. Both observe that due to certain properties in dynamic languages, such as first-class-types, design pattern implementations can become much simpler, up to the point of them disappearing, should they be captured by language constructs. The impact of certain paradigms upon design pattern implementation has also been analyzed, as done by Hannemann [4]. There the benefits of Aspect-Oriented programming with AspectJ is investigated with the result of substantial enhancement of the modularity of the implementations.

As static metaprogramming is a feature often present in dynamic programming languages, and can also be regarded as a more general form of AOP, the results are interesting when compared to the previous work. One interesting question for further investigation would be which of the actual benefits of the previously analyzed languages and techniques are in fact a direct result of the application of static metaprogramming techniques.

7.3 Classification of Design Patterns

In this paper, the classification of the patterns is based upon the similarities in communication between participating objects in a design pattern, in order to exploit possible specializations in the reusable implementations of the patterns by reusing parts of the communication protocol.

Previous work has been done to classify design patterns according to a broad range of characteristics:

Agerbo [10] classifies design patterns into fundamental patterns, which are not covered by programming languages, and language-dependent design patterns, whose implementation varies with the programming language used.

Another classification is provided by Gil [20], which is done by evaluating patterns by their distance from being actual language features as being either uses of cliches, which are defined to be common language features, idioms which are non-mainstream features of programming languages, or cadet patterns, which are not yet incorporated in programming languages. It is interesting to note that the main work performed in this paper could be viewed as an attempt to turn each design pattern into a part of the language, thus changing its classification.

8 Conclusion

In this article, we have demonstrated the benefits of using static metaprogramming techniques to alleviate some problems common to the implementation of design patterns.

This is done by using static metaprogramming to introduce new constructs that can be used by the pattern implementor similar to elements of the normal language. When using these constructs, the user specifies metadata that provides information about the concrete problem he wishes to solve, using idioms that correspond to the design pattern he wishes to use. By using this approach, the intent of the programmer is made explicit, thereby documenting the approach taken and alleviating the tracing problem.

As shown, the benefit of the implementations of the patterns as metaprograms directly depends on the additional information that can be concluded from or extracted with the help of the metadata supplied by the user. Often, the additional data can directly be devised from the protocol implied by the pattern itself. The more derived data or implied protocol is present in a pattern, the more effective the implementation is in saving work. Due to the approach taken, this method is also, in principle, capable of combining several patterns in one object, if care is taken during the implementation of the metaprogram.

Other conclusions presented include the benefits of using metaprograms to generate code specific for the design patterns, as one can eliminate design decisions in favor of more static type safety while at the same time preventing errors resulting from simple omissions in the manual transcription work.

Another important conclusion is the benefit of grouping patterns by their similarity in protocol, or, more precisely, their method of extracting additional metadata. This grouping helps organize design patterns in a fashion more applicable to direct implementation when compared to the previous classifications. Of additional interest is the specialization hierarchy that can be imposed upon the design patterns, as it results in fewer design patterns as roots when compared with other classifications. With the help of this classification, a better understanding of the properties of particular programming languages or paradigms with respect to implementing patterns is also possible, as one can compare language features with metaprograms, as a language feature directly corresponds to a metaprogram.

However, this classification is purely based upon the structure of the implementation, which can be regarded as being orthogonal to the usual classifications

which are based upon the intent which leads to the use of a particular design pattern.

On a more concrete side, the actual implementations of the patterns done in Common Lisp with CLOS should be regarded as a proof-of-concept, as for the sake of simplicity some options and desirable features have been omitted.

References

1. Baumgartner, G., Läufer, K., Russo, V.F.: On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University (1998)
2. Norvig, P.: Design patterns in dynamic programming. (1996)
3. Sullivan, G.T.: Advanced programming language features for executable design patterns. Technical Report AIM-2002-005, MIT Artificial Intelligence Laboratory (2002)
4. Hannemann, J., Kiczales, G.: Design pattern implementation in java and AspectJ. In Norris, C., Fenwick, J.J.B., eds.: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02). Volume 37, 11 of ACM SIGPLAN Notices., New York, ACM Press (2002) 161–173
5. Sheard, T.: Accomplishments and research challenges in meta-programming. Lecture Notes in Computer Science **2196** (2001) 2–44
6. Kiczales, G., Rivieres, J.D., Bobrow, D.: The Art of the Metaobject Protocol. MIT Press, Cambridge, MA (1991)
7. Bryant, A., Catton, A., Volder, K.D., Murphy, G.C.: Explicit programming. In: Proceedings of the 1st international conference on Aspect-oriented software development, ACM Press (2002) 10–18
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY (1995)
9. Kuehne, T.: Internal iteration externalized. In Guerraoui, R., ed.: Proceedings ECOOP '99. Volume 1628 of LNCS., Lisbon, Portugal, Springer-Verlag (1999) 329–350
10. Agerbo, E., Cornils, A.: How to preserve the benefits of design patterns. In: Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98). Volume 33, 10 of ACM SIGPLAN Notices., New York, ACM Press (1998) 134–143
11. Lieberherr, K., Orleans, D., Ovlinger, J.: Aspect-oriented programming with adaptive methods. Communications of the ACM **44** (2001) 39–41
12. Agerbo, E., Cornils, A.: Theory of language support for design patterns. Master's thesis, Department of Computer Science, Aarhus University (1997)
13. Soukop, J.: Implementing patterns. In Coplien, J.O., Schmidt, D.C., eds.: Pattern Languages of Program Design. Addison-Wesley Publishing Company (1995) 395–412
14. Tokuda, L., Batory, D.: Evolving object-oriented designs with refactorings. In: 14th IEEE International Conference on Automated Software Engineering, IEEE Computer Society Press (1999) 174–182

15. Tokuda, L., Batory, D.: Automating three modes of evolution for object-oriented software architectures. In: Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99), Berkeley, CA, USENIX Association (1999) 189–202
16. Budinsky, F., Finnie, M., Vlissides, J., Yu, P.: Automatic code generation from design patterns. IBM Systems Journal **35** (1996) 151–171
17. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley Publishing Company (2001)
18. Vlissides, J., Alexandrescu, A.: To code or not to code. C++ Report **March/June** (2000)
19. Bosch, J.: Design patterns as language constructs. Journal of Object-Oriented Programming **11** (1998) 18–32
20. Gil, J., Lorenz, D.H.: Design patterns vs. language design. In Bosch, J., Mitchell, S., eds.: Object-Oriented Technology: ECOOP'97 Workshop Reader. Volume 1357 of Lecture Notes in Computer Science., Springer (1997) 108–111 Workshop on Language Support for Design Patterns and Frameworks.